Infrastructure view

2024-10-08

**Last modified** : 2024-10-08

**Last global review** : *<perform regular global reviews of the view at least once a year while the project is active and mention the date here>*

**Document status** : *<'WIP', 'DRAFT', …​>*

# Introduction

This is the infrastructure point of view of the application. It describes the deployment of application modules in production and all technical components involved.

The other views of the document are accessible [from here](./README.adoc). The project glossary is available [here](glossary.adoc). We will not redefine the functional or technical terms used here.

This point of view deals with the infrastructure: servers, networks, operating systems, databases, middlewares, etc.

In short, it covers everything that is external to the application modules but necessary for their execution.

## Reference Documentation

Mention here the reference (defined at a IS level) architecture documents. This file should never summarize their content under penalty of quickly becoming obsolete and unmaintainable.

Documentary references

| N ° | Version | Document title / URL | Detail |
| --- | --- | --- | --- |
| *1* |  | *backup\_rules.pdf* | *Backup rules* |

# Not ruled

## Points subject to further study

Points subject to further study

| ID | Detail | Status | Subject holder | Deadline |
| --- | --- | --- | --- | --- |
| *EI1* | *The technical choice of the API Management solution remains subject to further study* | *WIP* | *SRE team* | *BEFORE 2040* |

## Assumptions

Assumptions

| ID | Detail |
| --- | --- |
| *HI1* | *We assume that by the time the project will be released, PostgreSQL 12 will be validated internally.* |

# Constraints

Constraints are the limits applicable to the requirements on the project.

It is interesting to explain them in order to obtain realistic requirements. For example, it would not be valid to require an availability incompatible with the Tier security level of the data center that will host it.

## Constraints on availability

The elements provided here can serve as a basis for the SLO (Service Level Objective). Ideally, this file should simply point to such an SLO without further clarification. When available, it may be augmented with others metrics like MTTF (Mean Time Between Failures).

This chapter has a pedagogical vocation because it highlights the maximum possible availability: the final availability of the application can only be lower.

### MTTD

Provide here the elements which make possible to estimate the average incident detection time.

Example 1: hypervision is done 24/7/365

Example 2: the production support service is available during office hours but an on-call duty is set up with alerting by e-mail and SMS 24/7 from Monday to Friday.

### Monitoring tools

Give here the tools and monitoring rules imposed at the IS level and any related constraints.

Example 1: The application will be supervised using Zabbix

Example 2: The batches must be able to be launch using a REST endpoint

Example 3: A failed batch must not be scheduled again without a human acknowledgment

### MTTR

Provide the elements to estimate the average repair time (Mean Time To Repair). Note that it is important to distinguish MTTD from MTTR. Indeed, it is not because a fault is detected that the skills or resources necessary for its correction are available.

Specify the time slots for operators to be present during the day and the possibilities of on-call duty.

If you have statistics or post-mortems, mention the average effective durations already observed.

List here the intervention times of the hardware, software, electricity, telecom service providers, etc.

We tentatively divide this section into "Hardware", "System and Virtualization", "Network", and "Data Recovery" subsections. Other categories are possible.

#### Hardware

Describe here the elements used to predict the MTTR of hardware elements (servers / racks / network equipment / electrical systems, etc.). List for example here the durations of intervention of the material service providers, electricity….

Example 1: Five spare physical servers are available at any given time.

Example 2: The Hitashi support contract provides for an intervention on the SAN bays in less than 24 hours.

Example 3: Replacement of IBM hardware support on BladeCenter blades is provided in 4 hours from 8 am to 5 pm, working days only.

#### System and virtualization

List here the elements allowing to estimate the correction time of a problem related to the OS or to a possible virtualization solution.

Example 1: At least one expert from each main domain (system and virtualization, storage, network) is present during office hours.

Example 2: Like any application hosted at datacenter X, the application will have the presence of operators from 7 a.m. to 8 p.m. working days. No standby engineer is planned.

Example 3: The observed restore time of a 40 GiB VM Veeam backup is 45 mins.

#### Network

List here the elements related to the network allowing to estimate the durations of intervention of the service providers or Telecom suppliers…​

Example 1: A network engineer is on call every weekend.

Example 2: Orange’s SLA provides for restoration of the Internet connection under nominal conditions in less than 24 hours.

#### Data Restore

List here the elements allowing to evaluate the duration of data restoration (files / objects / database). The RTO requirements listed below should take this MTTR into account.

Example 1: The Barman restore time of a Postgresql database is approximately (in hours) 0.1\*x + 0.2\*y with x, the size of the database in GiB and y the number of days of logs to replay.

Example 2: Restoring an offline backup (on tape) requires at least 4 hours of additional preparation.

### Scheduled interruptions

Give here the list and the duration of the standard programmed interruptions (maintenance windows).

Example 1: We estimate the interruption for maintenance of each server at 5 mins per month. The base effective server availability rate is therefore 99.99%.

Example 2: following security updates to certain RPM packages (kernel, libc, etc.), the RHEL servers are restarted automatically the night of the Wednesday following the update. This will result in an downtime of 5 mins on average 4 times a year.

### Level of service of the datacenter

Give here the security level of the data center (DC) according to the Uptime Institute scale (Tier from 1 to 4).

It should be noted that modern Cloud architectures favor the redundancy of DCs across distant sites rather than a higher Tier level at a single site (provided that data can be replicated effectively and a delay in immediate data consistency is acceptable, see the CAP theorem). Simplistically, it can be calculated that the availability of two active DCs in parallel is seven nines versus four nines for a Tier 4 DC. A compromise between the two models is deployment in redundant zones of the same site, at the cost of greater vulnerability to disasters.

Tier levels of data centers (source: Wikipedia)

| Tier level | Features | Availability rate | Annual statistical unavailability | Hot maintenance possible? | Fault-tolerance? |
| --- | --- | --- | --- | --- | --- |
| Tier 1 | Not redundant | 99.671% | 28.8 h | No | No |
| Tier 2 | Partial redundancy | 99.749% | 10 p.m. | No | No |
| Tier 3 | Maintainability | 99.982% | 1.6 hrs | Yes | No |
| Tier 4 | Fault tolerance | 99.995% | 0.4 h | Yes | Yes |

Example: the Madrid DC is Tier 3

### Summary of floor availability

Taking into account the previous elements, estimate the maximum floor availability of an application module (excluding disaster). Any requirement should be lower than this. In the case of a cloud, use the supplier’s SLA as a basis. In the case of an internally hosted application, take into account the availability of the datacenter and scheduled downtimes.

Example: <datacenter availability> \* <% of time not in scheduled maintenance> \* <system availability> \* <hardware availability> = 99.8 x 99.99 x 99.6 x 99.99 = ~ \* 99.4% \*.

### Disaster management (DRP/BCP)

A Disaster Recovery Plan (DRP) contains IT procedures and systems allowing IT services to be resumed ASAP after a disaster. DRP is a subset of a Business Continuity Plan (BCP). BCP provides an holistic perspective of the business procedures and systems required for an organization to continue in case of a disaster. A DRP focus on the IT part of it.

Disaster Management is a complex subject. In most cases, it is managed at an IS level. It is one of the strengths of public clouds (GCP, AWS, Azure…​) to handle a part of this complexity for you. Specific offers exist: see Disaster Recovery as a Service (DRaaS).

Disasters can be classified into three categories :

* Natural (earthquakes, floods, hurricanes, hot weather…​).
* Incident in the datacenter (accidental like industrial accidents, fires, major electrical failures, major network/storage/compute hardware outages, major sysadmin errors or intensional: military, terroristic, sabotage…​).
* Cyber: DDOS, virus, ransomware…​

Some BCP leverage High Availability (HA) architectures to allow continuity of critical IT activities of the organization during a disaster without notable interruption. Basically, a DRP focus on how to restore an IS after a disaster while HA architecture focus on making it work even when a disaster occurs.

The most important requirements to take into account when designing the architecture are the **RPO** (Recovery Point Objective, i.e. how much data we agree to lose since last backup) and the **RTO** (Recovery Time Objective, i.e. the maximum acceptable time to resume the operations). The lower the RTO and RPO, the more associated costs increase. It is therefore important to choose the right architecture for each IT service according to its importance and budget. An HA architecture targets a RTO and a RPO of zero or very near zero.

IT architects have two main options:

* When targeting a near zero RTO, only an HA architecture (like a multi-zones active-active clusters) can meet the requirement.
* For less demanding RTO (from several hours to several days), the most important thing is the time of data download and restoration into a backup DC.

Both options requires either an alternate site (at least ~10 km away from the main DC) or a public cloud solution. Note that synchronous data replication between DC is realistic only for short distances (few kms). For more distant DC, the latency is too high for most use cases. Asynchronous replication can be used at the price of loosing a few seconds of transactions when an incident occurs.

Describe among others:

* Redundant equipment in the second data center, number of spare servers, capacity of the standby data center compared to the main data center.
* Recovery measures (OS, data, applications).
* Organization’s RTO and RPO.
* Data replication mode between DC.
* Failback policy (reversibility): should we switch back to the first datacenter? How ?
* How are the blank tests organized? How often?

Example: As a reminder (see[doc xyz]), the VMs are replicated in the standby alternative datacenter via vSphere Metro Storage Cluster technology using SRDF in asynchronous mode. In the event of a disaster, the replicated VM at the standby site are up to date and ready to start.

Example 2: Two spare servers are available in the London site. Business data is backed-up every 4 hours and uploaded to BackBlaze.com. Estimated RPO is therefore 4H and RTO 2H.

## Hosting

* Where will this application modules be hosted? "on premises" datacenter? Private cloud? IaaS? PaaS? other?
* Who will operate this application modules? internally? Outsourced? No administration at all (PaaS) …​?

Example 1: This application will be hosted internally in the NYC datacenter (the sole to ensure the required service availability) and will be operated by the Boston team.

Example 2: Given the very high level of security required to run the application, the solution should only be operated internally by sworn officials. For the same reason, cloud solutions are excluded.

Example 3: Given the very large number of calls from this application to the PERSON repository, both will be collocated in the XYZ VLAN.

## Network constraints

List the constraints dealing with the network, in particular the theoretical maximum bandwith and the divisions into security zones.

Example 1: the LAN has a maximum bandwith of 10 Gbps

Example 2: The application components of intranet applications must be located in a trusted zone that cannot be accessed from the Internet.

## Deployment constraints

List the constraints related to the deployment of applications and technical components.

Example 1: A Virtual Machine should only host a single Postgresql instance.

Example 2: Java applications must be deployed as an executable jar and not as a war.

Example 3: Any application must be packaged as an OCI image and deployable on Kubernetes via a set of structured manifests in Kustomize format.

## Log constraints

List the constraints related to logs

Example 1: an application must not produce more than 1 Tio of logs/month.

Example 2: the maximum retention period for logs is 3 months.

## Backup and restore constraints

List the constraints related to backups

A common constraint is adherence to the 3-2-1 method:

* At least 3 copies of the data (the active data + 2 backups);
* At least 2 different storage technologies for these 3 copies (example: SSD for the active data and two backups on tape);
* At least 1 offline and offsite copy (example: a set of tapes stored in a fireproof safe at the bank).

Example 1: The maximum disk space that can be provisioned by a project for backups is 100 TiB.

Example 2: the maximum retention period for backups is two years

Example 3: Count 1 min/GiB for a NetBackup restore.

## Costs

List the budget limits.

Example 1: AWS Cloud service charges should not exceed $5K/year for this project.

# Requirements

Contrary to the constraints which fixed the boundaries to which any application had to conform, the non-functional requirements are given by the project decision-makers.

Schedule interviews to collect requirements. To result into something useful, interviews must be educational, recall the constraints and highlight realistic costs.

If certain requirements are still not realistic, mention it in the "Points subject to further study" section.

## Operating ranges

The main operating ranges are listed here (do not go into too much detail, this is not a production plan).

Think about users located in other time zones.

The information given here will be used as input to the application SLA.

Example of operating windows

| No window | Hours | Detail |
| --- | --- | --- |
| *1* | *From 8:00 a.m. to 7:30 p.m. NYC time, 5 days/7 working days* | *Intranet users* |
| *2* | *9:00 p.m. to 5:00 a.m. NYC time* | *Batches running* |
| *3* | *24/7/365* | *Internet users* |

## Availability requirements

We list the availability requirements here. The technical measures to achieve them will be given in the technical architecture of the solution.

These information can be used as input to the application **SLA**.

Be careful to frame these requirements because decision-makers often tends to request very high availability without always realizing the implications. The cost and complexity of the solution increases exponentially with the level of availability required.

The physical, technical or even software architecture can be completely different depending on the availability requirements (middleware or even database clusters, expensive hardware redundancies, asynchronous architecture, session caches, failover, etc.).

It is generally estimated that high availability (HA) starts at two new ones (99%), that is to say around 90 hours of downtime per year.

Give the availability requested by range.

The availability required here must be consistent with the “Constraints on availability” of the IS.

Maximum allowable downtime per range

| Operation range ID | Maximum downtime |
| --- | --- |
| *1* | *24h, maximum 7 times a year* |
| *2* | *4 hours, 8 times a year* |
| *3* | *4 hours, 8 times a year* |

## Degraded modes

Specify the degraded application modes.

Example 1: The *mysite.com* site must be able to continue to accept orders in the absence of the logistics department.

Example 2: If the SMTP server no longer works, the emails will be stored in the database and then resubmitted following a manual operation by the operators.

## Robustness requirements

The robustness of the system indicates its ability not to produce errors during exceptional events such as overload or failure of one of its components.

This robustness is expressed in absolute value per unit of time: number of (technical) errors per month, number of messages lost per year, etc.

Be careful not to be too demanding on this point because great robustness can imply the implementation of fault-tolerant systems that are complex, expensive and that can go against the capacity to scale up, or even availability.

Example 1: no more than 0.001% of requests in error

Example 2: the user must not lose his shopping cart even in the event of a breakdown (be careful, this type of requirement impacts the architecture in depth, see the [Availability](#X015299be265b9faea39774cb4e24450bf081441) section).

Example 3: the system should be able to withstand a load three times greater than the average with a response time of less than 10 seconds at the 95th percentile.

## RPO requirements

Give here the Recovery Point Objective (RPO) of the application (i.e. how much data we agree to lose since last backup) in unit of times.

Data restoration occurs mainly in following cases:

* Hardware data loss (unlikely with redundant systems).
* A power-user or operator error (quite common).
* An application bug.
* A deliberate destruction of data (ransomware-type attack) …​

Example: We shouldn’t loose more than one working day of application data.

## RTO Requirements

The Recovery Time Objective (in unit of times) is the maximum authorized time objective for reopening the service following an incident.

This requirement must be compatible (less than or equal) to the MTTR given in constraint above. It is in fact useless to require an RTO of 1H if the operators have measured an effective MTTR of 2H. It must also be compatible with the availability requirement.

Specify this value only to clarify a precise restoration objective, otherwise, do not complete this section and refer to the MTTR constraint above.

Example: We must be able to restore and put back online the 3 TiB of the XYZ database in 1 hour maximum.

## Deployment and update requirements

### Server side

Specify here how the application should be deployed on the server side.

For example :

* Is the installation manual? scripted with IT Automation tools like Ansible or SaltStack? via Docker images?
* How are the components deployed? As packages? Are we using a package repository (type yum or apt)? Do we use containers?
* How are the updates applied?

### Client side

Specify here how the application should be deployed on the client side:

* If the application is large (large .js files or images for example), is there a risk of an impact on the network?
* Local proxy caching to be expected?
* Are firewall rules to be expected?
* (For a Java application): which version of JRE is needed on clients?
* (For a standalone application): which version of the OS is supported?
* If the OS is Windows, does the installation go through a deployment tool (Novell ZENWorks for example)? Does the application come with a Nullsoft-style installer? Does it affect the system (environment variables, registry, etc.) or is it in portable mode (single zip)?
* If the OS is Linux, should the application be provided as a package?
* How are the updates applied?

### Specific deployment strategy

* Are we planning a blue/green deployment?
* Are we planning a canary testing type deployment? if so, on what criteria?
* Are we using feature flags? if so, on which features?

Example: The application will be deployed in blue/green mode. Once ready, a DNS switch will point to machines with the new version.

## Concurrency management requirements

Specify the internal or external components that may interfere with the application.

Example 1: All components of this application must be able to run concurrently. In particular, batches/GUI concurrency must always be possible because the batches must be able to run during the day.

Example 2: Batch X should only be started if batch Y is completed correctly, otherwise data may be corrupted.

## Ecodesign requirements

Ecodesign consists of limiting the environmental impact of the software and hardware used by the application. Requirements in this area are generally expressed in WH or CO2 equivalent.

Also take into account impressions.

Check out the EPA’s Greenhouse Gas Equivalencies Calculator for CO2/KWH equivalency.

Example 1: The Power usage effectiveness (PUE) of the site must be 1.5 or less.

Example 2: Ink and paper consumption should be reduced by 10% compared to 2020.

# Target architecture

## Principles

What are the main infrastructure principles of our application?

Examples:

* Application components exposed to the Internet are located in a DMZ protected behind a firewall then a reverse-proxy.
* Regarding interactions between the DMZ and the intranet, a firewall only allows communications from the intranet to the DMZ.
* Active/active clusters will be exposed behind an LVS + Keepalived with direct routing for the return.

## Availability

Availability represents the minimum proportion of time a system over a year during which it works in acceptable conditions. It is expressed as a percentage (example: 99.9%).

List here the measures taken to meet the availability requirements. The available measures are very varied and should be chosen by the architect according to their respective contribution and cost.

We can group availability measures into four main categories:

* **Monitoring measures** allowing to detect faults as early as possible, hence lowering the MTDT (average detection time).
* **Organizational measures**:
  + Human presence (on-call, extended support hours, etc.) which improves the MTTR (average resolution time) and without which monitoring is inefficient;
  + Quality of incident management (see ITIL best practices), for example, is an Incident Management Procedure written? Direct enough (for instance several hierarchical validations decrease the MTTR)?
* **High Availability (HA)** measures like clusters or RAID.
* **Data Recovery measures**: is the recovery procedure well defined? Tested? Being able to quickly restore the last backup greatly improves the MTTR.

**Availability and redundancy**:

* The **availability of a set of serial components** can be computed by this formula: A = A1 \* A2 \* …​ \* An. Example: the availability of an application using a Tomcat server available at 98% and an Oracle database available at 99% would be 97.02%.
* The **availability of a set of components in parallel** can be computed by this formula: A = 1 - (1-A1) \* (1-A2) \* .. \* (1-An). Example: the availability of three clustered Nginx servers each available at 98% is 99.999%.
* It is important to be consistent on the **availability of each link** in the linking chain: there is no point in having an expensive active/active cluster of JEE application servers if all these servers call a database located on a single server physical with disks without RAID.
* A system is estimated to be **highly available (HA) from 99%** availability.
* The term **“spare”** designates a spare device (server, disk, electronic card, etc.) which is dedicated to the need for availability but which is not activated outside of failures. Depending on the level of availability requirement, it can be dedicated to the application or shared at the IS level.
* The main **redundancy models** (NMR = N-Modular Redundancy) are listed below (with N, the number of devices ensuring correct operation under load and that we can replicate):
  + **N**: no redundancy (example: when a server single power supply fails, the server is down)
  + **N+1**: a spare component is available (but not yet active), we can support the failure of a piece of equipment (example: we have a spare power supply available).
  + **N+M**: a single spare component cannot handle the load, at least M spare devices are required.
  + **2N**: the system is fully redundant and active and can withstand the loss of half of the components (example: we have two power supplies, if one fails, the server keeps running). This system is considered Fault-Tolerant.
  + **2N+1**: In addition to a fully mirrored system, a backup system is available (for maintenance operations for instance).

**Clustering**:

* A cluster is a **set of nodes (servers) hosting the same application module**.
* Depending on the level of availability sought, each node can be:
  + **active**: the node processes the requests (example: one Apache web server among ten and behind a load balancer). Failover time: zero;
  + **passive in “hot standby” mode**: the node is installed and started but does not process requests (example: a MySQL slave database which becomes master). MTTR: a few seconds (failure detection time);
  + **passive in “warm standby” mode**: the node is started and the application is installed but not started (example: a server with a turned off Tomcat instance hosting our application). In case of failure, the application is started automatically. MTTR: of the order of a minute (time for detection of the failure and activation of the application);
  + **passive in "cold standby" mode**: the node is a simple spare. To use it, we have to install the application, configure and start it. MTTR: from tens of minutes with virtualization solutions (eg: KVM live migration) and/or containers (Docker) to several hours on systems where none automatic deployment features are available.
* There are two active/active cluster architectures:
  + **Loosely coupled active/active clusters** in which one node is completely independent from the others, either because the application is stateless (the best case), or because the context data (typically an HTTP session) is managed in isolation by each node. In the latter case, the load balancer must ensure session affinity, i.e., always route requests from a client to the same node and in the event of failure of this node, the users routed there lose their session data and need to reconnect (Note: the nodes all share the same data persisted in the database, the context data on each node is only transient data in memory).
  + **Strongly coupled active/active clusters** in which all nodes share the same data. In this architecture, all context data must be replicated in every node (e.g. distributed cache of HTTP sessions replicated with JGroups).

**Failover**:

Failover is the ability of a cluster to ensure that in the event of a failure, requests are no longer sent to the failed node but to a running node. This process is **automatic**.

Without failover, it is up to the client to detect the failure and reconfigure itself to only call the running nodes. In fact, this is rarely practicable and the **clusters almost always have failover capacities**.

A failover solution can be described by the following attributes:

* Which **Failover strategy** ? For instance: "Fail fast" (a node is considered as down as soon as a failure is detected), "On fail, try next one", "On fail, try all".
* Which **fault detection solution**?
  + Load balancers can use a wide variety of health checks (mock requests, CPU analysis, logs, etc.) to check the nodes they control;
  + Active/passive clusters failure detections work most of the time by listening to the heartbeat of the active server by the passive server, for example via UDP multicast requests in the VRRP protocol used by keepalived.
* How long does it take to detect the failure? failure detection solutions should be configured correctly (as short as possible without degradation of performance) to limit the duration of the failover.
* What **relevance of the detection**? is the down server \* really \* down? a bad setting or a network micro-cut should not cause a total unavailability of a cluster while the nodes are still healthy.
* What strategy for **failback**?
  + in an **N-to-1** cluster, we will failback on the server which had broken down once repaired and the failed server will become the backup server again;
  + in an **N-to-N** cluster (an architecture in the process of democratization with the PaaS type cloud like AWS Lambda or CaaS like Kubernetes): the services previously running on the failed node are distributed to the remaining nodes (the cluster having been sized in anticipation of this possible overload).
* **Transparent for the caller or not**? In general, the requests pointing to a server whose failure has not yet been detected fall in error (in timeout most of the time). Some advanced Fault Tolerant systems or architectures can make it transparent for the client.

A few words about **load balancers**:

* A load balancer (**) is a \*mandatory brick** for an active/active cluster.
* In the case of clusters, a classic error is to make LB a **SPOF**. We would then reduce the total availability of the system instead of improving it. When dealing with the clusters with a availability vocation (i.e. not only performance-oriented), it is necessary to redundant the LB itself in active/passive mode (obviously not in active/active mode otherwise, we would only shift the problem). The passive LB must monitor the active LB at high frequency and replace it automatically as soon as it falls.
* It is crucial to configure correctly and at a sufficient frequency the **heath checks** to the destination nodes because otherwise the LB will continue to send requests to failed or overloaded nodes.
* Some advanced LBs (example: redispatch option of HAProxy) allow the failover process to be transparent seen from the client by retrying to other nodes in the event of a failure or timeout and therefore improve fault tolerance since we avoid to return an error to the caller during the fault pre-detection period.
* **Round Robin load repartition algorithm is not always the best choice**. A simple algorithm is the LC (Least Connection) allowing the LB to favor the least loaded nodes. Other clever algorithms exist and can be taken into consideration (weight systems per node or combination load + weight for example). However, make sure to carefully test and understand the chosen algorithm implications to avoid any catastrophic outage.
* In the Open-Source world, see for example LVS + keepalived or HAProxy + keepalived.

**Fault tolerance**:

Fault Tolerance (FT = Fault Tolerance) should not be confused with HA; It is **stricter version of HA where availability is 100% and no data can be lost** (Wikipedia: "Fault tolerance is the property that enables a system to continue operating properly in the event of the failure of (or one or more faults within) some of its components"). Historically, it meant a full hardware redundancy. In a micro-services world, it can also be achieved at a software level with active-active clusters. Moreover, a true fault-tolerance system should avoid significant performance degradation seen by the end-users.

For example, a RAID 1 drive provides transparent fault tolerance: in case of failure, the process writes or reads without error after the automatic failover to the healthy disk. A Kubernetes cluster can achieve fault tolerance as well by starting new PODs. Or a clustered in-memory distributed cache can avoid losing any HTTP session.

To allow fault tolerance of a cluster, it is essential to have an active/active cluster with strong coupling in which the **context data is replicated at all times**. Another (much better) solution is to simply avoid context data (by keeping session data in the browser via a JavaScript client for example) or to store it in database (SQL/NoSQL) or in distributed ans synchronously replicated cache (at a cost on performances).

To get fully transparent fault tolerance, it is also necessary to use a **load balancer able to make retries by itself**.

Do not take lightly a FT requirement because in general these solutions:

* Makes the **architecture not only more expensive but also more complex** and therefore can make it less robust and more expensive to build, test, operate. Only mission or life-critical softwares usually need it.
* **Can degrade performance**: Availability and performance solutions are generally linked (for example, a cluster of stateless machines will divide the load by the number of nodes and at the same time, the availability increases), but sometimes, availability and performance can be antagonistic: in the case of a stateful architecture, typically managing HTTP sessions with a distributed cache (like Infinispan replicated in synchronous mode or REDIS with persistence on the master), any transactional update of the session adds an additional cost linked to updating and replicating caches. If one of the nodes crashes, the user keeps his session at the next request and does not have to reconnect, but the cost is high.
* **Can even degrade the global availability** because all nodes are strongly coupled and synchronized. A software update for example can force the shutdown of the entire cluster.

Some availability solutions

| Solution | Cost | Implementation complexity (indicative) | Availability level gain (indicative) |
| --- | --- | --- | --- |
| Disks in RAID 1 | XXX | X | XXX |
| Disks in RAID 5 | X | X | XX |
| Redundancy of power supplies and other components | XX | X | XX |
| Ethernet card bonding | XX | X | X |
| Active / passive cluster | XX | XX | XX |
| Active / active cluster with LB | XXX | XXX | XXX |
| Servers / spare hardware | XX | X | XX |
| Good system monitoring | X | X | XX |
| Good application monitoring | XX | XX | XX |
| Remote uptime checks | X | X | XX |
| On call staff dedicated to the application, 24/7/365 | XXX | XX | XXX |
| Copy of the backup of the last working-day database dump on SAN bay (for express restoration) | XX | X | XX |

Example 1: To achieve the required 98% availability, the envisaged availability measures are as follows:

* All servers in RAID 5 + redundant power supplies.
* HAProxy + keepalived active/passive LB shared with other applications.
* Active / active cluster of two Apache + mod\_php servers.
* Spare server that can be used to rebuild the MariaDB database from the backup of the day before in less than 2 hours.

Example 2: To achieve the required availability of 99.97%, the availability mechanisms considered are as follows:

* Application hosted in a Tier 3 DC.
* All servers in RAID 1 + redundant power supplies + bonding interfaces.
* HAProxy + keepalived active/passive LB dedicated to the application.
* Active/active cluster of four servers (i.e., 2N redundancy) Apache + mod\_php.
* Oracle instance in RAC on two machines (with dedicated FC interconnection).

## Deployment in production

Provide here the deployment model in the target environment on the various middleware and physical nodes (servers). Represent network equipment (firewalls, appliances, routers, etc.) only if they help understanding.

Naturally, it will be preferably documented with a UML2 deployment diagram or a C4 deployment diagram.

For clusters, give the instantiation factor of each node.

Comment out if necessary the affinity constraints (two components must run on the same node or the same middleware) or anti-affinity (if two components must not run on the same node or in the same middleware).

Clearly identify the hardware dedicated to the application.

Example:

## Versions of infrastructure components

List here OS, databases, MOM, application servers, etc …​

Example of infrastructure components

| Component | Role | Version | Technical environment |
| --- | --- | --- | --- |
| *CFT* | *Secure file transfer* | *X.Y.Z* | *RHEL 6* |
| *Wildfly* | *JEE application server* | *9* | *Debian 8, OpenJDK 1.8.0\_144* |
| *Tomcat* | *Web container for UIs* | *7.0.3* | *CentOS 7, Sun JDK 1.8.0\_144* |
| *Nginx* | *Web server* | *1.11.4* | *Debian 8* |
| *PHP + php5-fpm* | *Dynamic pages of the XYZ GUI* | *5.6.29* | *nginx* |
| *PostgreSQL* | *RDBMS* | *9.3.15* | *CentOS 7* |

## Matrix of network flows

List here all the network flows used by the application. Listening ports should be specified. We also detail here the protocols (JMX or SNMP for example).

In some organizations, this matrix will be too detailed for an architecture document and will be kept in a document managed by the integrators or the operators.

It is not necessary to refer to application flows here because readers are not looking for the same information. Operators or integrators seek completeness of network flows to configure infrastructure (like firewalls).

Include useful information about the network being used in order to assess the performance (TR, latency) and security: LAN, VLAN, Internet, LS, WAN, …​)

Partial example of a network flows matrix

| ID | Source | Destination | Network type | Protocol | Listening port |
| --- | --- | --- | --- | --- | --- |
| *1* | *lb2* | *IP multicast 224.0.0.18* | *LAN* | *VRRP over UDP* | *3222* |
| *2* | *lb1* | *host1, host2* | *LAN* | *HTTP* | *80* |
| *3* | *host3, host4, host5* | *bdd1* | *LAN* | *PG* | *5432* |
| *4* | *sup1* | *host[1-6]* | *LAN* | *SNMP* | *199* |

## Environments

Give here an overall view of the environments used by the application. The most common environments are: development, testing, acceptance, pre-production/benchmarks, production, training.

It is often useful to subdivide environnements into 'platforms' made up of a set of technical components isolated from each other (although they may share common hardware resources). For example, a test environment can consist of lanes UAT1 and` UAT2` allowing two testers to work in isolation.

Environments

| Environment | Role | Content | Nb of platforms |
| --- | --- | --- | --- |
| Development | Continuous deployment (CD) for developers | Develop branch deployed on each commit | 1 |
| Acceptance | UAT | Tag deployed at the end of each Sprint | 2 (UAT1 and UAT2) |

## Ecodesign

List here the infrastructure measures to meet the "Ecodesign Requirements".

The solutions are often the same as those to performance requirements. In this case, just refer to it.

However, ecodesign analyzes and solutions can be specific to this theme. Some tips for improving energy performance:

* Measure the electrical consumption of the systems with the [PowerAPI](http://www.powerapi.org/) probes (developed by INRIA and the University of Lille 1).
* Use caches (opcode cache, memory caches, HTTP caches …​).
* Use containers to greatly optimize VMs or physical machines usage (especially when dealing with RAM consumption).
* Host servers in a high-performance datacenter. Cloud providers generally offer more efficient data centers than on-premises ones. The unit of comparison here is the PUE (Power Usage Effectiveness), a ratio between the energy consumed by the data center and the energy actually used by the servers (therefore excluding cooling and external devices). Some DC can achieve PUE as low as 1.2 against 2.5 on average (2017).
* However :
  + verify the origin of the energy (see for example the analyzes of Greenpeace in 2017 on [the use of energy from coal and nuclear](http://www.clickclean.org) by Amazon);
  + keep in mind that the energy consumed by the application on the client and network side is much greater than that used on the server side (for example, we can estimate that a server consuming barely more than one workstation is enough for several thousands or even tens of thousands of users). Energy reduction also involves extending the lifespan of terminals and the use of soberer devices.

Example 1: setting up a Varnish cache in front of our CMS will reduce the number of PHP dynamic page construction by 50% and will save two servers.

Example 2: The application will be hosted on a cloud with a PUE of 1.2 and an 80% renewable source of electrical energy.

## Load regulation

### Circuit breakers

In some cases, extreme and unpredictable peaks are possible (flash crowd).

If this risk is identified, provide a fuse system with offset of all or part of the load on a static website with an error message for example.

This measure can also be used in the event of a DDOS-type attack because it allows already connected users to finish their transactions properly.

### Quality of Service

It is also useful to provide dynamic application regulation systems, for example:

* Via throttling (set a maximum number of requests by origin and unit of time).
* Token systems (which also make it possible to favor some clients over others by granting them a greater number of tokens).

Example 1: The total number of tokens for calls to REST operations on the DetailArticle resource will be 1000. Beyond 1000 simultaneous calls, callers will get an unavailability error 429 that they will have to manage (and possibly make retries in exponential backoff mode).

Example 1: distribution of tokens will be as follows by default:

| Operation on DetailArticle | Proportion of tokens |
| --- | --- |
| GET | 80% |
| POST | 5% |
| PUT | 15% |

Example 2: a throttling of 100 requests per source and per minute will be set up in the reverse proxy.

## Timeout management

In general, all distributed calls (in particular HTTP(S) to APIs or object storage and calls to databases) must be limited in connection time AND execution time. Without these timeouts, deadly module contentions can occur in the event of performance issues.

Describe here the different timeouts implemented on the linking chains. Keep in mind that from client to persistence, timeouts should decrease as you go through the linking chain (example: 10 secs on the Reverse proxy, 8 secs on the REST endpoint, 5 secs on the database). In the opposite case, a technical component can continue to process a request when its calling component has already given up, which poses both problems of wasting resources but above all effects that are difficult to predict.

Also avoid using exactly the same value in all the technical components to avoid unexpected effects linked to the concomitant timeouts.

Example:

| Component | Timeout (ms) |
| --- | --- |
| Rest JavaScript Client | 5000 |
| API Gateway | 4000 |
| API Rest Node.js | 3500 |
| PG database | 3000 |

## Operations

List here the main operating principles of the solution. The details (saved filesystems, production plan, processing planning …​) will be recorded in separate documents.

If this application remains in the organization’s standard, simply refer to any common document.

### Stop/start-up sequence

Specify the starting and the stopping sequence of devices and components required to run the application modules. Depending on the situation, you can include some external components or not.

Some others operations documents will contain a more precise version of this chapter (for instance using detailed SystemD "Wants"), only describe here the general principles.

Starting sequence is generally done in the reverse direction of the linking chains and stopping in the direction of the linking chain. Most of the time, the stopping sequence is simply the reversed sequence version of the starting sequence.

Specify any issues in the event of a partial start-up sequence (for example, will the application server connection pool retry to connect to the database if it is not yet started? How many times?)

Example of a start-up sequence:

1. pg1 on bdd1 server
2. mq1 on bdd1
3. services1 on host3, host4 and host5 servers
4. services2 on host3, host4 and host5 servers
5. batches on servers host1, host2
6. GUI on servers host1, host2

Example of stopping sequence:

1. GUI on servers host1, host2
2. batches on servers host1, host2
3. services2 on host3, host4 and host5 servers
4. services1 on host3, host4 and host5 servers
5. mq1 on bdd1
6. pg1 on bdd1 server

### Scheduled operations

List macroscopically (don’t try to make it exhaustive, we just need jobs streams principles applying):

* The jobs or family of jobs and their possible inter-dependencies. Specify if a scheduler will be used.
* Internal tasks (cleaning/checking tasks) of the system which only fulfill technical roles (purges, rebuilding indexes, deleting temporary data, etc.)

Example 1: the process-demand job will be launched every 5 mins by JobScheduler.

Example 2: the internal ti\_index process is a Java class calling REINDEX commands in JDBC. It is launched from a Quartz scheduler once a month.

### Switch to maintenance mode procedure

Explain (if necessary) the measures and procedures allowing to put the application offline explicitly for the users.

Example: We will use the F5 BigIp LTM to display an unavailability page.

### Backups and restores

Give the general safeguard policy. It must meet the "RPO Requirements". Likewise, restoration measures must be compatible with the "Availability Requirements":

* Are backups hot ? Cold ?
* Which data should be saved? (carefully select the data to be backed up because the cumulative volume of the backups set can easily reach ten times the backed up volume).
  + system images / snapshots for server or VM recovery?
  + full filesystems or directories?
  + databases in dump format? binary format?
  + the logs? traces ?
* Are the backups encrypted? if so, specify the encryption algorithm used and how the key will be managed.
* Are the backups compressed? if so, with which algorithm? (lzw, deflate, lzma …​)? using which compression level ? be careful to find the compromise between compression/decompression time and storage gain.
* What tools are used? (simple cron?, Quartz, "backup-manager" tool? IBM TSM?).
* What technology is used for backups? (LTO or DLT magnetic tapes? external drives? RDX cartridges? cloud storage like Amazon S3? optical support? NAS? …​)
* What is the frequency of each type of backup? (do not go into too much detail here)
* What is the backup strategy?
  + complete? incremental? differential? (Take into account availability requirements. Restoring an incremental backup will take longer than a differential backup restore, itself longer than a full backup restore);
  + which rollover strategy?
* Backup execution report: content, how is it sent? where are the logs?
* Where are the backups stored? (ideally as far as possible from the backed up system while allowing restoration in a time compatible with availability requirements).
* Who has physical access to the backups and their logs? to the encryption key? (think about confidentiality requirements).

We recommend :

* to use a medium separate from the source data;
* to have at least two separate storage media if data is critical to the organization (e.g. hard disk + magnetic tape);
* to make sure that the backups are not modifiable by the machine which was backed up (for example, a backup on NAS may be deleted by mistake at the same time as the backed up data);
* to use read-only features with online backups systems to deal with ransomwares (some online backups systems have a temporary read-only attribute feature).

Rollover example: set of 21 backups over one year:

* 6 daily incremental backups;
* 1 full backup on sunday which serves as a weekly backup;
* 3 weekly backups corresponding to the 3 other sundays. The support of the last sunday of the month becomes the monthly backup;
* 11 monthly backups corresponding to the last 11 months.

Finally, it’s important to keep in mind that what we *really* want are restores, not backups. It is crucial to ensure that the restoration will be functional:

* Are the backups valid and complete?
* What restoration tests are planned? how often (once a year is a minimum)?
* How long will a restoration (benchmarks) take? Is it compatible with the RTO?
* Do we have external dependencies that can slow us down (safe accessible during the day only, for example)?
* Do we have enough resources for the restoration (intermediate storage, CPU and memory for decompression/decryption, etc…​)?

### Logs

Without being exhaustive on the log files, present the general policy for the production and management of logs:

* What are the log turnover policies? is the rollover managed by the application (via a DailyRollingFileAppender log4j for example) or by the system (typically via the logrotate daemon)?
* Is a centralization of logs planned? (essential for SOA or micro-services architectures). See for example the ELK stack.
* What is the level of verbosity expected by type of component? we have often to choose between the WARN and INFO levels. If the developers have used the INFO level for relevant information (environment at startup for example) and not the DEBUG, set the INFO level.
* Are anti-log injection (XSS) measures taken?

Example 1: the application logs of the service-allmydata component will be in production at INFO level with daily rotation and kept two-month.

Example 2: the logs will be escaped using the Jakarta commons-lang StringEscapeUtils.escapeHtml() method.

### Monitoring

Monitoring is a central pillar of availability by drastically reducing MTTD (average failure detection time).

Ideally, it will not only be reactive but also proactive (weak signals detection).

Metrics are raw measurements (% CPU, FS size, size of a pool, etc.) from system, middleware or application probes.

Indicators are logical combinations of several metrics with thresholds (e.g. 'critical level if the CPU usage on server s1 remains above 95% for more than 5 minutes').

#### Technical monitoring

List the metrics:

* System (% of file system usage, load, swap in/out volume, number of threads…​)
* Middleware (% of HEAP used on a JVM, number of threads on the JVM,% use of a pool of threads or JDBC connections ..)

Example: we measure the percentage of wait-io and the server load.

#### Application monitoring

List the application metrics (developed internally). They can be technical or functional:

* Number of requests to access a screen.
* Number of contracts processed per hour.
* …​

It is also possible to set up BAM (Business Activity Monitoring) tools based on these metrics to monitor process-oriented indicators.

Example: the application monitoring REST API will offer a Metric resource containing the main business metrics: number of packages to send, number of active preparers, etc.

#### Monitoring tools

Such tools (like Nagios, Zabbix, Hyperic HQ in the Open Source world):

* Collect metrics (in SNMP, JMX, HTTP …​) periodically.
* Persist metrics in some type of time series database (like RRD).
* Consolidates indicators from metrics.
* Displays trends over time for these indicators.
* Allows setting alert thresholds based on indicators and notifying operators in the event of exceeding.

Example: the management of the monitoring will be based on the Grafana platform.

#### Alerting

Specify here the alert conditions and the channel used

Example: an SMS is sent if no request occurs for the last 4 hours or if the number of technical errors of a component exceeds 10/h.

#### Black box monitoring

It is also highly desirable and inexpensive to provide a black box monitoring test system (via scenarios run automatically). The idea here is to test a system as a whole and with the most external end-user view possible (unlike whitebox monitoring for which specific components are supervised).

In general, these tests are simple (HTTP requests from a scheduled curl for example). They must be launched from one or more remote sites to detect network cuts.

In general, they don’t perform update actions but only read-only ones. If they perform updates, it will be necessary to be able to identify in all the components the data resulting from this type of requests in order not to pollute the business data and the decision-making systems. We don’t advice such tests because they bring a lot of complexity.

Example for a website: black box monitoring tests will be implemented via HTTP requests launched via the uptrends.com tool. In the event of a breakdown, an email is sent to the operators.

#### Metrology

Are we monitoring the performance of the application in production?

This allows:

* To have factual feedback on *in vivo* performance and to improve the quality of decisions about possible resizing of the hardware platform.
* To proactively detect failures (following a sudden drop in the number of requests for example).
* Perform statistical analysis on the use of components or services in order to promote decision-making (for the decommissioning of an application, for example).

There are three main groups of solutions:

* APMs (Application Performance Monitoring): tools that inject probes without application impact, which collect and restore them (some even reconstitute the complete linking chains via correlation identifiers injected during distributed calls). Example: Oracle Enterprise Manager, Oracle Mission Control, Radware, BMC APM, Dynatrace, Pinpoint in Open-Source …​). Check that the overhead of these solutions is still negligible or limited and that the stability of the application is not jeopardized.
* In-house metrology by logs if the requirements are low.
* External query sites which periodically call the application and produce dashboards. They have the advantage of taking into account the WAN times not available via internal tools. To be used in conjunction with black-box monitoring (see above).

Example: site performance will be continuously monitored by pingdom.com. More in-depth performance analyzes will be implemented by Pinpoint as needed.

## Jobs scheduling

Indicate the scheduler or planner used to manage the jobs and consolidate the production plan (example: VTOM, JobScheduler, Dollar Universe, Control-M, …​). Detail any specificities of the application:

* Degree of parallelism of jobs
* Mandatory time slots
* Retries in case of error
* …​

Do the jobs have to produce an execution report? in what form and with what content?

Example 1: the jobs will be scheduled by the JobScheduler instance of the organization.

* The jobs should never run on public holidays.
* Their execution will be limited to the periods 11h00pm-06h00am. Their schedule must therefore be in this range or they will not be launched.
* We will not launch more than five instances of the job J1 in parallel.

Example 2: The jobs will have to produce an execution report at each launch (with basic data such as the number of elements processed, the duration of the treatment and any relevant indicator).

## Migrations

This chapter describes a required migrations from older systems.

Describe on a macroscopic scale the planned procedure as well as the rollback strategy in case of problem.

Describe, if necessary, a dry-run operation in parallel with the old system.

Example 1: The X component will be replaced by the Y services. Then the Z Oracle database will be migrated in one-shot via a PL/SQL + DBLink script to the X instance with the new basic format of the T component.

Example 2: In the event of a problem with the new component, a rollback will be provided: the old data will be restored within two hours and the new data from the failover will be taken over by the S1 script.

## Decommissioning

This chapter will be read when the application reaches the end of its life and must be removed or replaced.

Among other things, he describes:

* Data to be archived or on the contrary destroyed with a high level of confidence.
* The physical components to be removed or destroyed.
* Server and/or client side uninstallation procedures (it is common to see obsolete components still running on servers and causing performance and security issues that go under the radar).
* Security constraints associated with decommissioning (this is a sensitive step that is often overlooked, for example hard drives can be found filled with very sensitive data following a donation of equipment).

Example: The X, Y, and Z servers will be transferred to the social service for charitable donation after completely erasing the hard drives using the shred command, three passes.